GOODRICH / TAMASSIA

ESTRUCTURAS DE DATOS Y ALGORITMOS EN JAVA

2A. EDICIÓN

CECSA

Capítulo

4 Pilas, colas y colas dobles

Contenido

Las colas y pilas son las estructuras de datos más sencillas de todas, pero además se consideran las más importantes. Las pilas y las colas se emplean en diferentes aplicaciones que incluyen estructuras más complicadas de datos. Además, las pilas y las colas se encuentran entre las pocas clases de estructuras de datos que se implementan con frecuencia en las microinstrucciones del hardware dentro de un CPU. Éstas son básicas para algunas funciones importantes de los ambientes de cómputo moderno, como por ejemplo el ambiente del tiempo de ejecución de Java llamado Máquina Virtual Java.

En este capítulo se definen, en forma general, los tipos de datos abstractos pila y cola, y se presentan dos implementaciones alternativas de ellos: arreglos y listas enlazadas. Para ilustrar la utilidad de las pilas y las colas, se presentan ejemplos de su aplicación a ejecuciones en la máquina virtual Java. También, se presenta una generalización de pilas y colas llamada cola doble o cola de doble terminación, y se indica cómo implementarla con una lista doblemente enlazada. Además, en este capítulo se incluyen descripciones de algunos conceptos de programación como interfaces, casting, centinelas y el patrón adaptador. El capítulo termina con un caso de estudio donde se usa la estructura de datos de pila, para formar una sencilla aplicación de análisis de acciones de bolsa.

4.1 Pilas

Una ***pila*** es un depósito, o *contenedor* de objetos que se insertan y sacan de acuerdo con el principio **LIFO: *último en entrar, primero en salir*** (*last-in, first-out*). En cualquier momento se pueden insertar objetos en una pila, pero en cualquier momento sólo se puede sacar el objeto que se insertó más recientemente (esto es, el último). El nombre "pila" se debe a la metáfora o comparación de una pila de platos de un servidor de platos de resorte en una cafetería. En este caso, las operaciones fundamentales son el "empuje" de los platos para meterlos en la pila, y "quitar" cuando se sacan. Cuando sea necesario sacar un nuevo plato del despachador, se hace "quitar" el plato superior, y cuando se agrega un plato, se "empuja" hacia abajo para hacerle un lugar en la pila, convirtiéndose en el plato superior. Quizá una comparación más divertida sería un despachador PEZ® de dulces, que guarda pastillas de menta en un contenedor con acción de resorte, que hace saltar la pastilla superior de la pila cuando se levanta su tapa (véase la figura 4.1). Las pilas son estructuras de datos fundamentales que se usan en muchas aplicaciones, entre las que se encuentran:

Ejemplo 4.1: *Los navegadores de Internet guardan en una pila las direcciones de los sitios recién visitados. Cada vez que un usuario visita un sitio nuevo, su dirección se "empuja" para meterla en la pila de direcciones. Después, el navegador permite que el usuario "quite" el sitio recién visitado, mediante el botón "atrás".*

Ejemplo 4.2: *Los editores de texto suelen tener una función "deshacer" (undo) que cancela las operaciones recientes de edición, y hace regresar al documento a sus estados anteriores. Esta operación de deshacer se logra guardando los cambios de texto en una pila.*

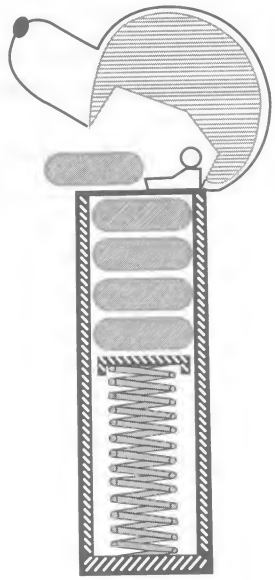

Figura 4.1: Esquema de un despachador PEZ®; es una implementación física del TDA pila. PEZ® es una marca registrada de PEZ Candy, Inc.

4.1.1 Tipo de dato abstracto pila

Una pila S es un tipo de dato abstracto (TDA) que soporta los dos métodos fundamentales siguientes:

push(o): Insertar el objeto (o) en la parte superior de la pila.
Entrada: objeto; ***Salida:*** ninguna.

pop(): Sacar el objeto superior de la pila y regresarlo; se produce un error si la pila está vacía.
Entrada: ninguna; ***Salida:*** objeto.

Además, también se definirán los siguientes métodos de soporte:

size(): Regresa la cantidad de objetos en la pila.
Entrada: ninguna; ***Salida:*** entero.

isEmpty(): Regresa un valor booleano que indica si la pila está vacía.
Entrada: ninguna; ***Salida:*** booleana.

top(): Regresa el objeto superior de la pila sin sacarlo de ella; se produce un error si la pila está vacía.
Entrada: ninguna; ***Salida:*** objeto.

Ejemplo 4.3: *La siguiente tabla muestra una serie de operaciones de la pila y sus efectos sobre una pila S inicialmente vacía de objetos enteros. Para simplificar se usan enteros, en lugar de objetos enteros como argumentos de las operaciones.*

Operación	***Salida***	***S***
push(5)	–	(5)
push(3)	–	(5, 3)
pop( )	*3*	(5)
push(7)	–	(5, 7)
pop( )	*7*	(5)
top( )	*5*	(5)
pop( )	*5*	( )
pop( )	"*error*"	( )
isEmpty( )	*verdadero*	( )
push(9)	–	(9)
push(7)	–	(9, 7)
push(3)	–	(9, 7, 3)
push(5)	–	(9, 7, 3, 5)
size( )	*4*	(9, 7, 3, 5)
pop( )	*5*	(9, 7, 3)
push(8)	–	(9, 7, 3, 8)
pop( )	*8*	(9, 7, 3)
pop( )	*3*	(9, 7, 3)

Interfaz de pila en Java

Por su importancia, la estructura de datos pila se incluye como clase "constructora" en el paquete java.util de Java. La clase java.util.Pila es una estructura de datos que guarda objetos genéricos de Java que incluyen, entre otros, los métodos push(obj), pop(), peek() (equivalent to top()), size() y empty() (equivalent to isEmpty()). Los métodos pop() y peek() producen la excepción StackEmptyException si se les llama con una pila vacía. Si bien es conveniente usar sólo la clase constructora java.util.Stack, es bueno aprender cómo diseñar e implementar una pila "partiendo de cero".

La implementación de un tipo de dato abstracto en Java se hace en dos pasos. El primero es la definición de una ***interfaz de programación de aplicación*** Java (API, de *Application Programming Interface*), o ***interfaz*** solamente, que describe los nombres de los métodos que soporta el TDA, y cómo se deben declarar y usar. En el fragmento de programa 4.1 se muestra una interfaz completa de Java para el TDA pila. Nótese que esta interfaz es muy general, porque especifica que en la pila se pueden insertar objetos de clases arbitrarias y posiblemente heterogéneas.

```
/**
 * Interfaz para una pila: una colección de objetos
 * que se insertan y se sacan de acuerdo con el
 * principio último en entrar primero en salir.
 *
 * @autor Roberto Tamassia
 * @autor Michael Goodrich
 * @véase StackEmptyException
 */
public interface Stack {
  /**
   * @regresa la cantidad de elementos en la pila.
   */
  public int size();
  /**
   * @regresa cierto si la pila está vacía; regresa falso en caso contrario.
   */
  public boolean isEmpty();
  /**
   * @regresa el elemento superior de la pila.
   * @excepción StackEmptyException si la pila está vacía.
   */
  public Object top()
    throws StackEmptyException;
  /**
   * Insertar un elemento en la parte superior de la pila.
   * @parám elemento es el elemento a insertar.
   */
  public void push (Object element);
  /**
   * Sacar el elemento superior de la pila.
   * @regresa el elemento sacado.
   * @excepción StackEmptyException si la pila está vacía.
   */
  public Object pop()
    throws StackEmptyException;
}
```

Fragmento de programa 4.1: Interfaz Stack documentada con comentarios en estilo Javadoc. (Véase la sección 1.9.2.)

La condición de error que sucede al llamar el método pop() o el método top() cuando la pila está vacía se señala lanzando una excepción del tipo StackEmptyException, que se define en el fragmento de programa 4.2.

```
/**
 * Excepción en tiempo de ejecución lanzada cuando se trata de hacer la
   operación superior o sacar en una pila vacía.
 */

public class StackEmptyException extends RuntimeException {
   public StackEmptyException(String err) {
      super(err);
   }
}
```

Fragmento de programa 4.2: Excepción lanzada por los métodos pop() y top() de la interfaz Stack cuando se llaman con la pila vacía.

Para que un TDA sea de utilidad hay que proporcionar una clase concreta que implemente los métodos de la interfaz asociada con ese TDA. En la siguiente subsección se indicará una implementación sencilla de la interfaz Stack.

4.1.2 Implementación sencilla basada en un arreglo

En esta subsección se mostrará cómo observar una pila, guardando sus elementos en un arreglo. Como se debe determinar el tamaño de un arreglo al momento de crearlo, uno de los detalles importantes de la implementación es especificar algún tamaño máximo N para la pila; por ejemplo, $N = 1000$ elementos. Entonces, la pila consiste en un arreglo S de N elementos más una variable entera t, que expresa el índice del elemento superior del arreglo S. (Figura 4.2.)

Figura 4.2: Observación de una pila mediante un arreglo S. El elemento de arriba en la pila se guarda en la celda $S[t]$.

Al considerar que los arreglos comienzan en Java con el índice 0, se inicializa t en -1, y se usa este valor de t para indicar cuándo está vacía la pila. De igual forma, se puede usar esta variable para determinar la cantidad de elementos $(t + 1)$ en una pila. También se introducirá un nuevo tipo de excepción, llamada StackFull Exception para indicar la condición de error que se produce si se trata de insertar un elemento nuevo y la pila S se encuentra llena. La excepción StackFullException es específica de esta implementación de una pila, y no se define en el TDA de la pila. Dada esta excepción nueva, se pueden entonces implementar los métodos TDA de la pila, como se describe en el fragmento de programa 4.3.

```
Algoritmo size():
    return t + 1

Algoritmo isEmpty():
    return (t < 0)

Algoritmo top():
    if isEmpty() then
        throw a StackEmptyException
    return S[t]

Algoritmo push(o):
    if size() = N then
        throw a StackEmptyException
    t ← t + 1
    S[t] ← o

Algoritmo top():
    if isEmpty() then
        throw a StackEmptyException
    e ← S[t].
    S[t] ← null
    t ← t − 1
    return e
```

Fragmento de programa 4.3: Implementación de una pila mediante un arreglo.

Lo correcto de esos métodos es consecuencia inmediata de su misma definición. Sin embargo, hay un punto de cierto interés en esta implementación, que implica poner en funcionamiento el método pop. Nótese que se pudo haber evitado restablecer el $S[t]$ anterior a **null** y el método seguiría siendo correcto. Sin embargo, hay un compromiso por eliminar esta asignación en Java. Ese compromiso implica el mecanismo ***recolección de basura*** de Java, que busca en la memoria objetos que ya no son referenciados por objetos activos, y recupera el espacio que ocupan para uso en el futuro. Sea $e = S[t]$ el elemento superior antes de llamar al método pop. Si se hace que $S[t]$ sea una referencia nula, se indica que la pila ya no necesita guardar un objeto de referencia e. En realidad, si no hay otras referencias activas a e, el espacio de memoria ocupado por e será recuperado por el recolector de basura. Una implementación concreta en Java de la especificación anterior en pseudocódigo, mediante la clase Java ArrayStack que implementa la interfaz Stack, se muestra en los fragmentos de programa 4.4 y 4.5.

```
/**
```

```
 * Implementación de la interfaz Stack mediante un arreglo de longitud fija.
 * Se lanza una excepción si se intenta ejecutar una operación de meter
   cuando el tamaño de la pila es igual a la longitud del arreglo.
 *
 * @autor Natasha Gelfand
 * @autor Roberto Tamassia
 * @véase StackFullException
 */
public class ArrayStack implements Stack {
  /**
   * Longitud predeterminada del arreglo usado para implementar la pila.
   */
   public static final int CAPACITY = 1000;
  /**
   * Longitud del arreglo que se usa para implementar la pila.
   */
   private int capacity;
  /**
   *Arreglo usado para implementar la pila.
   */
   private Object S[];
  /**
   * Índice del elemento superior de la pila en el arreglo.
   */
   private int top = -1;
  /**
   * Inicializar la pila para usar un arreglo de longitud predeterminada
     CAPACIDAD.
   */
   public ArrayStack() {
      this(CAPACITY);
   }
  /**
   * Inicializar la pila para usar un arreglo de longitud dada.
   *
   * @parám cap, longitud de la pila.
   */
   public ArrayStack(int cap) {
      capacity = cap;
      S = new Object[capacity];
   }
```

Fragmento de programa 4.4: Implementación basada en arreglo de Java, de la interfaz Stack (continúa en el fragmento de programa 4.5).

```
  /**
```

```
    * Tiempo O(1)
    */
    public int size() {
        return (top + 1);
    }
    /**
    * Tiempo O(1).
    */
    public boolean isEmpty() {
        return (top < 0);
    }
    /**
    * Tiempo O(1).
    * @excepción StackFullException si el arreglo está lleno.
    */
    public void push(Object obj) throws StackFullException {
        if (size() == capacity)
            throw new StackFullException("Desbordamiento de pila.");
        S[++top] = obj;
    }
    /**
    * Tiempo O(1).
    */
    public Object top() throws StackEmptyException {
        if (isEmpty())
            throw new StackEmptyException("La pila está vacía.");
        return S[top];
        }
    /**
    * Tiempo O(1).
    */
    public Object top() throws StackEmptyException {
        Object elem;
        if (isEmpty())
            throw new StackEmptyException("La pila está vacía.");
        elem = S[stop];
        S[top--] = null; // quitar referencia S[superior] para recolección de basura.
        return elem;
    }
}
```

Fragmento de programa 4.5: Implementación en Java, basada en arreglo, de la interfaz Stack. (Continúa del fragmento de programa 4.4.) Nótese que los comentarios Javadoc de los métodos de la interfaz Stack sólo mencionan información específica de la clase (por ejemplo, el tiempo de ejecución del método) que todavía no se incluye en el fragmento de programa 4.1.

La tabla 4.1 muestra los tiempos de ejecución para métodos en una realización de una pila mediante un arreglo. Cada uno de los métodos de la pila de la realización del arreglo ejecuta una cantidad constante de instrucciones de programa donde intervienen operaciones aritméticas, comparaciones y asignaciones. Por consiguiente, en esta implementación del TDA Pila, cada método se ejecuta en tiempo constante; es decir, cada uno se ejecuta en el tiempo $O(1)$.

Método	Tiempo
size	$O(1)$
isEmpty	$O(1)$
top	$O(1)$
push	$O(1)$
pop	$O(1)$

Tabla 4.1: Ejecución de una pila realizada mediante un arreglo. El uso de espacio es $O(N)$, siendo N el tamaño del arreglo, determinado cuando se instancia la pila. Nótese que el uso del espacio es independiente de la cantidad $n \leq N$ de elementos que estén en realidad en la pila.

La implementación de una pila con un arreglo es sencilla y eficiente a la vez, y se usa en una diversidad de aplicaciones de cómputo. Sin embargo, esa implementación tiene un aspecto negativo: debe asumir una cota superior fija N del tamaño máximo de la pila. En el fragmento de programa 4.4 se escogió el valor de capacidad $N = 1000$, en forma más o menos arbitraria. En realidad, en una aplicación se podría necesitar mucho menos espacio, y en ese caso se desperdiciaría la memoria. También se podría necesitar más espacio en una aplicación, en cuyo caso esta implementación como pila podría "destruir" la aplicación con un error tan pronto como trate de empujar en la pila el objeto $(N + 1)$. Así, aun con su simplicidad y su eficiencia, la implementación de pila basada en arreglo no necesariamente es lo ideal. Por fortuna hay otras implementaciones, que se describirán más adelante en este capítulo, que no tienen limitación de tamaño y usan un espacio proporcional a la cantidad real de elementos guardados en la pila. Sin embargo, en los casos en que se cuenta con un buen estimado de la cantidad de elementos que deben dirigirse a la pila, es difícil ganarle a la implementación basada en arreglo. Las pilas tienen un papel fundamental en varias aplicaciones de cómputo, por lo que es útil tener una implementación TDA rápida de la pila, como por ejemplo la sencilla basada en un arreglo.

Casting con una pila genérica

Una de las grandes ventajas de tener en cuenta la pila TDA mediante una clase Java que implementa la interfaz Stack es que se pueden guardar objetos genéricos en ella, siendo cada uno de éstos de una clase arbitraria. (Véanse de nuevo los fragmentos

de programa 4.1 y 4.4-4.5.) Es decir, un ArrayStack puede guardar objetos Integer, Student o hasta Planet.

Sin embargo, hay que tener en cuenta que se debe ser consistente en cómo usar la pila genérica porque todos los elementos que se guardan en ella se consideran instancias de la clase Object de Java. Esto no es problema cuando se agregan elementos a la pila, puesto que toda clase de Java hereda de la clase Object. Sin embargo, al recuperar un objeto de la pila (sea con el objeto superior o el método sacar), siempre se obtiene una referencia del tipo Object, sin importar cuál fue la clase específica del objeto. Así, para usar el elemento recuperado como una instancia de la clase específica a la que pertenece en realidad, se debe hacer un ***cast***, que obliga a que el objeto se considere como miembro de una clase específica, y no como una superclase Object más general. En la sección 2.5 se describe con más detalle el casting en Java.

En el ejemplo del fragmento de programa 4.6 se ilustrará la necesidad del casting en una aplicación sencilla que usa una pila.

```
public static Integer[] reverse(Integer[] a) {
   ArrayStack S = new ArrayStack(a.length);
   Integer[] b = new Integer[a.length];
   for (int i = 0; i < a.length; i++)
      S.push(a[i]);
   for (int i=0; i < a.length; i++)
      b[i] = (Integer) (S.pop());
   return b;
}
```

Fragmento de programa 4.6: Un método que invierte el orden de los elementos en un arreglo, usando un ArrayStack auxiliar. Se hace casting para obligar a que el objeto regresado por el método pop se considere como objeto Integer. Nótese que un arreglo Java tiene un campo length que guarda el tamaño del arreglo.

El método reverse del fragmento de programa 4.6 ilustra una pequeña aplicación de la estructura de datos pila, pero esa estructura tiene numerosas aplicaciones más importantes que ésta. De hecho, la estructura de datos pila juega un papel importante en la implementación del mismo lenguaje Java.

4.1.3 Pilas en la máquina virtual de Java

Un programa en Java se suele compilar en una secuencia de códigos de bytes que se definen como instrucciones "de máquina", en un modelo bien definido de máquina, la ***máquina virtual de Java***. La definición de esa máquina virtual es el corazón de la definición del mismo lenguaje Java. Al compilar un programa Java en códigos de byte para la máquina virtual Java, y no en el lenguaje de máquina de un

CPU específico, un programa Java puede ejecutarse en cualquier computadora, como una PC o una estación UNIX que puedan emular la máquina virtual de Java. Es interesante ver que la estructura de datos pila juega un papel central en la definición de esa máquina virtual.

La pila de métodos Java

Las pilas son aplicaciones importantes en el ambiente de ejecución de los programas Java. Un programa Java en ejecución (con más precisión: un hilo Java en ejecución) tiene una pila privada, llamada ***pila de métodos Java***, o simplemente ***pila Java***, que se usa para rastrear las variables locales y otra información importante sobre los métodos, a medida que se invocan durante la ejecución (véase la figura 4.3).

En forma más específica, durante la ejecución de un programa Java, la máquina virtual Java mantiene una pila cuyos elementos son descriptores de las invocaciones de métodos con validez en ese momento (esto es, invocaciones no terminadas). A esos descriptores se les llama ***frames***. Un *frame* (marco) para alguna invocación del método "enfriar" guarda los valores vigentes de las variables locales y parámetros del método enfriar, y también la información sobre el método que llamó enfriar, y sobre qué debe regresarse a este método.

La máquina virtual Java mantiene la dirección del paso de programa que ejecuta al momento, en un registro especial, llamado ***contador del programa***. Cuando un método "enfriar" invoca a otro método "engañar", el valor vigente del contador de programa se anota en el marco de la invocación actual de frío, para que la máquina virtual de Java sepa dónde regresar cuando se haya ejecutado el método engaño. Arriba de la pila Java está el marco del ***método de ejecución***, esto es, el método con mayor vigencia en el control de la ejecución. Los demás elementos de la pila son marcos de los ***métodos suspendidos***, que son los métodos que han invocado a otro método y que en la actualidad se encuentran en espera de que se les regrese el control al terminar. El orden de los elementos en la pila corresponde a la cadena de invocaciones de los métodos activos. Cuando se invoca un nuevo método, se mete a la pila un marco para este método. Al terminar, su marco se saca de la pila y la máquina virtual de Java reasume el procesamiento del método que previamente se haya suspendido.

La pila Java hace el pase de parámetro a los métodos. En forma específica, Java usa el protocolo de pase de parámetros denominado ***llamada por valor***. Eso quiere decir que el ***valor*** vigente de una variable (o una expresión) es lo que se pasa como argumento a un método que se haya llamado.

En el caso de una variable *x* de un tipo primitivo como int o float, el valor actual de *x* no es más que el número que está asociado con *x*. Cuando ese valor se pasa al método llamado, se asigna a una variable local en el marco de ese método. Esta sencilla asignación también se ilustra en la figura 4.3. Nótese que si el método llamado cambia el valor de esta variable local, ***no*** cambiará el valor de la variable en el método de llamada.

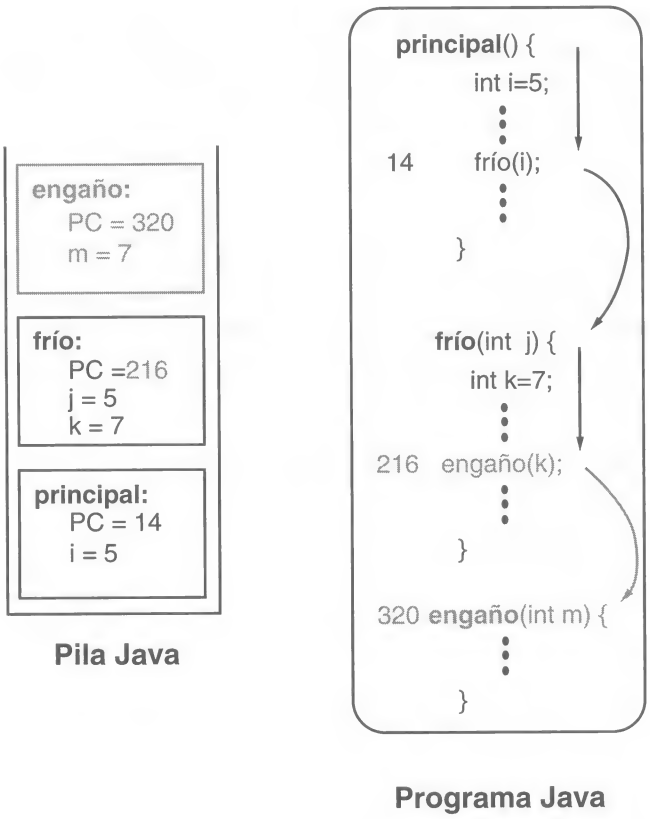

Figura 4.3: Ejemplo de una pila Java: el método engaño acaba de ser llamado por el método frío, que previamente fue llamado por el método principal. Nótense los valores del contador de programa, de los parámetros y de las variables locales guardados en los marcos de la pila. Cuando termina la invocación del método engaño, continúa la invocación del método frío, el cual reanuda su ejecución en la instrucción 217, que se obtiene incrementando el valor del contador de programa guardado en el marco de la pila.

Sin embargo, en el caso de una variable x que se refiera a un objeto, el valor actual de x es la dirección del objeto x en la memoria. (En la sección 4.2.3 se explica más acerca de dónde se encuentra en realidad dentro de la memoria.) Así, cuando se pasa un objeto x como parámetro a algún método, lo que sucede es que se pasa la dirección de x. Cuando se asigna esta dirección a alguna variable local y en el método llamado, y indicará el mismo objeto al que se refiere x. Por consiguiente, si el método llamado cambia el estado interno del objeto al que se refiere y, al mismo tiempo cambia el estado interno del objeto al que se refiere x, es decir, es el mismo objeto. Sin embargo, si el programa llamado cambia a y, para que se refiera a algún otro objeto, x quedará sin cambio; se referirá al objeto anterior.

Por lo anterior, la máquina virtual Java usa la pila de métodos para implementar llamadas de método y pase de parámetros. Por cierto, las pilas de métodos no son exclusivas de Java. Se usan en el ambiente de ejecución de los lenguajes más modernos de programación, incluyendo al C y al C++.

Recursión

Una de las ventajas de usar una pila para implementar la invocación de métodos es que permite usar en los programas la ***recursión***. Esto es, permite que un método se llame a sí mismo como subrutina. Por ejemplo, con la recursión se puede calcular la clásica función factorial, $n! = n(n-1)(n-2)\cdots 1$, como se muestra en el fragmento de programa 4.7.

```
public static long factorial(long n) {
   if (n < = 1)
      return 1;
   else
      return n*factorial(n–1);
}
```

Fragmento de programa 4.7: Método recursivo factorial.

El método factorial se llama a sí mismo en forma recursiva para calcular el factorial de $n-1$. Cuando termina la llamada recursiva, regresa $(n-1)!$, que a continuación se multiplica por n para calcular $n!$ A su vez, la invocación recursiva se llama a sí misma para calcular el factorial de $n-2$, etc. La cadena de invocaciones recursiva, y por consiguiente la pila Java, sólo crece hasta el tamaño n porque al llamar factorial(1) se regresa 1 de inmediato, sin invocarse a sí misma en forma recursiva. La pila Java en la máquina virtual Java permite que exista el método factorial en forma simultánea en varios marcos activos (hasta n en algún momento). Cada marco guarda el valor de su parámetro n, y también el valor a regresar.

Este ejemplo ilustra también una propiedad importante que siempre debería tener un proceso recursivo: que el método termine. En el método factorial se aseguró que fuera así, al escribir declaraciones no recursivas para el caso $n \leq 1$, y al hacer siempre la llamada recursiva a un valor menor del parámetro $(n-1)$ del que se dio (la n) para que, en algún punto (en el "fondo" de la recursión) se ejecute la parte no recursiva del cálculo (regresando 1). Siempre se debe diseñar un método recursivo en forma que se garantice su terminación en algún punto, por ejemplo, siempre con llamadas recursivas a instancias "menores" del problema, y manejando las instancias "mínimas" en forma no recursiva como casos especiales. Se observa que si se diseña un método "infinitamente recursivo", en realidad no funcionará para siempre. En lugar de ello, en algún punto agotará la memoria disponible para la pila Java y generará un error de falta de memoria. Sin embargo, si se usa con cuidado la recursión, la pila del método implementará métodos recursivos sin problema.

La recursión puede ser muy poderosa, ya que con frecuencia permite diseñar programas sencillos y eficientes para problemas bastante difíciles.

La pila de operandos

Es interesante ver que, en realidad, hay otro lugar donde la máquina virtual de Java usa una pila. Se usa una ***pila de operandos*** para evaluar expresiones aritméticas, como por ejemplo $((a + b)*(c + d))/e$. Una operación binaria simple, como $a + b$, se calcula al empujar una a hacia la pila de operandos, luego una b a la pila de operandos, y a continuación llamando a una instrucción que saque a los dos elementos de la pila de operandos, para realizar la operación binaria, y a continuación empuja el resultado de nuevo hacia la pila de operandos. De igual manera, las instrucciones para escribir y leer elementos a y de la memoria implican el uso de métodos pop y push para la pila de operandos.

En el capítulo 6 se describirá la evaluación de expresiones aritméticas en un panorama más general. Sin embargo, por ahora sólo se hará notar que la pila de operandos desempeña un papel básico en este cálculo. Por consiguiente, la pila de operandos y la pila Java (de métodos) son los componentes fundamentales de la máquina virtual de Java.

4.2 Colas

Otra estructura de datos fundamentales es la ***cola***. Es un "primo" cercano de la pila porque la cola es un contenedor de objetos que se insertan y se quitan de acuerdo con el principio ***FIFO*** (***first-in first-out***, primero que entra, primero que sale). Esto es, se pueden insertar elementos en cualquier momento, pero en cualquier momento sólo se puede sacar el elemento que haya durado más en la cola. Se dice que los elementos entran a la cola por ***detrás*** y salen por ***delante***. El modelo para esta terminología es una fila de espera de personas por entrar a un juego de un parque de diversiones. Las personas se forman al final de la cola y salen de ella en su principio.

4.2.1 Tipo de dato abstracto cola

El tipo de dato abstracto cola define, formalmente, un contenedor que guarda los objetos con un orden en el que se restringen el acceso y la salida de elementos al primer elemento de la secuencia, que se llama el ***frente*** de la cola, y la inserción de elementos se restringe al final de la secuencia, que se llama ***final*** de la cola. Esta restricción impone la regla de que los elementos se inserten y se saquen en una cola de acuerdo con el principio primero que entra, primero que sale (FIFO).

5.2 Listas

El uso de un rango no es el único medio de indicar el lugar en donde aparece un elemento en una lista. Si se tiene una lista S implementada con una lista enlazada (simple o doblemente), es posible que sea más natural y eficiente usar un ***nodo*** en lugar de un rango, como medio de identificar dónde se tiene acceso y actualizar una lista. En esta sección se explorará una forma de abstraer el concepto de "lugar" del nodo en una lista, sin revelar los detalles de cómo se implementa la lista.

5.2.1 Operaciones basadas en nodo

Sea S una lista lineal implementada con una lista doblemente enlazada. Se desea definir métodos para S que tomen como parámetros nodos de la lista y que muestren a nodos como tipos de retorno. Esos métodos podrían proporcionar aceleraciones importantes respecto a los métodos basados en rango ya que encontrar el rango de un elemento en una lista enlazada requiere buscar en toda la lista, en forma creciente, desde su inicio o término, contando los elementos en el proceso.

Por ejemplo, se podría definir un método removeAtNode(v) hipotético que saca el elemento de S guardado en el nodo v de la lista. El uso de un nodo como parámetro permite sacar a un elemento en el tiempo $O(1)$ sólo con ir directamente al lugar donde está guardado ese nodo, para a continuación "desenlazar" ese nodo mediante una actualización de los enlaces *siguiente* y *prev* de sus vecinos. De igual modo se podría insertar, en el tiempo $O(1)$, un nuevo elemento e en S con una operación como por ejemplo insertAfterNode(v,e), que especifica al nodo v después del cual hay que insertar el nuevo elemento. En este caso, tan sólo se "enlaza y mete" el nuevo nodo.

La definición de los métodos de un TDA lista, al agregar esas operaciones basadas en nodos, hace surgir el problema de cuánta información se debe exponer acerca de la implementación de la lista. Claro que es deseable usar una lista simple o doblemente enlazada, sin revelar ese detalle a un usuario. De igual forma no se quiere permitir que un usuario modifique la estructura interna de una lista sin que lo sepa el encargado. Sin embargo, esa modificación sería posible si se diera al usuario una referencia de un nodo de la lista, en una forma que le permitiera ingresar datos internos en ese nodo (como un campo *siguiente* o *prev*).

Para abstraer y unificar las formas distintas de guardar elementos en las diversas implementaciones de una lista se introduce el concepto de ***posición*** en una lista para que formalice la noción intuitiva del "lugar" de un elemento en relación con los demás de la lista.

5.2.2 Posiciones

Para ampliar con seguridad el conjunto de operaciones para listas, se abstrae una noción de "posición" que permita aprovechar la eficiencia de las implementaciones de lista simple o doblemente enlazada sin violar los principios del diseño orientado a objetos. En este marco, se considera a una lista como un contenedor de elementos que guarda a cada uno en una posición, y que mantiene arregladas esas posiciones en orden lineal. Una posición es, en sí misma, un tipo de dato abstracto que soporta el siguiente método sencillo:

element(): Regresa el elemento guardado en esta posición.
Entrada: ninguna; ***Salida:*** objeto.

Una posición se define siempre ***en forma relativa***, esto es, en función de sus vecinas. En una lista, una posición p siempre estará "después" de una posición q y "antes" de una posición s, a menos que p sea la primera o la última posición. Una posición p, que se asocia con cierto elemento e en una lista S, no cambia, aun cuando cambie el rango de e en S, a menos que en forma explícita se saque a e y con ello se destruya la posición p. Además, la posición p no cambia aun cuando se sustituya o se intercambie el elemento e guardado en p con otro elemento. Estos datos con las posiciones permiten definir un rico conjunto de métodos de lista basados en posición, que toman como parámetros los objetos posición, y también permiten que los objetos posición sean valores de retorno.

5.2.3 Tipo de dato abstracto Lista

Al aplicar el concepto de posición para encapsular la idea de "nodo" en una lista, se puede definir otra clase de TDA secuencia, llamada TDA ***lista***. Este TDA soporta los siguientes métodos para una lista S:

first(): Regresa la posición del primer elemento de S; se produce un error si S está vacío.
Entrada: ninguna; ***Salida:*** posición.

last(): Regresa la posición del último elemento de S; se produce un error si S está vacío.
Entrada: ninguna; ***Salida:*** posición.

isFirst(p): Regresa un valor booleano que indica si la posición dada es la primera en la lista.
Entrada: posición p; ***Salida:*** booleana.

isLast(p): Regresa un valor booleano que indica si la posición dada es la última en la lista.
Entrada: posición p; ***Salida:*** booleana.

before(p): Regresa la posición del elemento de S que precede la posición p; se produce un error si p es la primera posición.
Entrada: posición; ***Salida:*** posición.

after(p): Regresa la posición del elemento S que sigue al que está en la posición p; se produce un error si p es la última posición.
Entrada: posición; ***Salida:*** posición.

Los métodos anteriores permiten la referencia a posiciones relativas en una lista, partiendo del principio o del final, y para avanzar en forma ascendente o descendente en la lista. Se pueden imaginar, en forma intuitiva, esas posiciones como los nodos de la lista, pero nótese que no hay referencias específicas a objetos nodo, ni a sus enlaces *prev* y *siguiente* en esos métodos. Además de los métodos anteriores y de los métodos genéricos size y isEmpty, también se incluyen los siguientes métodos de actualización para el TDA lista.

replaceElement(p,e): Reemplaza con e al elemento de la posición p y regresa al elemento que antes ocupaba la posición p.
Entrada: Posición p y objeto e; ***Salida:*** objeto.

swapElements(p,q): Intercambia los elementos guardados en las posiciones p y q, de tal modo que el elemento de la posición p pasa a la posición q y el que estaba en la posición q pasa a la posición p.
Entrada: dos posiciones; ***Salida:*** ninguna.

insertFirst(e): Inserta un nuevo elemento e en S como primer elemento.
Entrada: objeto e; ***Salida:*** posición del elemento e recién insertado.

insertLast(e): Inserta un nuevo elemento e en S, como último elemento.
Entrada: objeto e; ***Salida:*** posición del elemento e recién insertado.

insertarBefore(p,e): Inserta un nuevo elemento e en S antes de la posición p en S; se produce un error si p es la primera posición.
Entrada: posición p y objeto e; ***Salida:*** posición del elemento e recién insertado.

insertarAfter(p,e): Inserta un nuevo elemento e en S, después de la posición p en S; se produce un error si p es la última posición.
Entrada: posición p y objeto e; ***Salida:*** posición del elemento e recién insertado.

remove(p): Quita al elemento de la posición p de S.
Entrada: posición: ***Salida:*** el elemento eliminado.

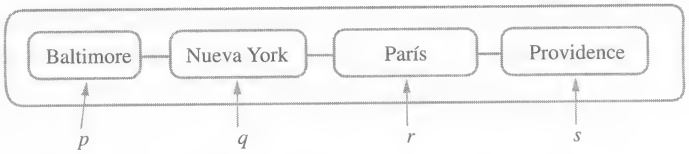

Figura 5.5: Ilustración de una lista. Las posiciones en el orden actual son *p*, *q*, *r* y *s*, Por ejemplo, *q* se define como la posición después de la posición *p* y antes de la posición *r*.

La lista permite considerar una colección ordenada de objetos en función de sus lugares, sin tener que preocuparse de la forma exacta en que se representan esos lugares (véase la figura 5.5).

También nótese que hay cierta redundancia en el repertorio anterior de operaciones para el TDA lista. Consiste en que se puede hacer la operación isFirst(*p*) viendo si *p* es igual a la posición mostrada por first(). De igual modo, la operación isLast(*p*) se reduce a llamar a last(). También se puede hacer la operación insertFirst(*e*) efectuando la operación insertBefore(first(), *e*) y la operación insertLast(*e*) se puede reemplazar con insertAfter(last(), *e*). Se puede considerar que los métodos redundantes son métodos abreviados para las operaciones comunes que contribuyen a la inteligibilidad del programa.

Obsérvese que se produce una condición de error si es inválida una posición pasada como argumento de alguna de las operaciones lista. Las razones para que *p* sea una posición inválida incluyen a:

- p = **null**
- *p* se había eliminado de la lista
- *p* es una posición de una lista distinta

Se usará una excepción llamada InvalidPositionException en estos casos.

En el siguiente ejemplo se ilustran las operaciones del TDA lista.

Ejemplo 5.4: *A continuación se muestra una serie de operaciones para una lista S inicialmente vacía. Se usan las variables* p_1, p_2, etc., *para indicar posiciones distintas, y se muestra entre paréntesis al objeto guardado en esa posición y en ese momento.*

Operación	***Salida***	***S***
insertFirst(8)	$p_1(8)$	(8)
insertAfter(p_1, 5)	$p_2(5)$	(8, 5)
insertBefore(p_2, 3)	$p_3(3)$	(8, 3, 5)
insertFirst(9)	$p_4(9)$	(9, 8, 3, 5)
before(p_3)	$p_1(8)$	(9, 8, 3, 5)
last( )	$p_2(5)$	(9, 8, 3, 5)
remove(p_4)	*9*	(8, 3, 5)
swapElements(p_1, p_2)	–	(5, 3, 8)
replaceElement(p_3, 7)	*3*	(5, 7, 8)
insertAfter(first( ), 2)	$p_5(2)$	(5, 2, 7, 8)

El TDA lista con su noción incorporada de posición, es útil en varios entornos. Por ejemplo, un programa que modele a varias personas jugando cartas podría describir en forma de lista la mano de cada persona. Como la mayoría de las personas gustan de juntar las cartas del mismo palo, la inserción y remoción de cartas de la mano de una persona se podría implementar con los métodos del TDA lista y las posiciones se determinan por un ordenamiento natural de los palos. De igual forma, un editor sencillo de texto tiene incorporada la noción de inserción y eliminación posicional porque se acostumbra hacer todas las actualizaciones en relación con un ***cursor***, que representa a la posición momentánea en la lista de caracteres de texto que se edita.

Hay varias formas de implementar el TDA lista. Es probable que la forma más natural y eficiente es usar una lista doblemente enlazada. A continuación se describirá cómo lograr esta implementación frecuente, aplicando los principios del diseño orientado a objetos.

5.2.4 Implementación de una lista doblemente enlazada

Se trata de implementar el TDA lista usando una lista doblemente enlazada. Se puede hacer que los nodos de la lista enlazada implementen el TDA posición. Esto es, se hace que cada nodo implemente la interfaz Position y en consecuencia se define un método element(), que regresa al elemento guardado en el nodo. De este modo, los nodos mismos funcionan como posiciones. La lista enlazada los considera al interior como nodos, pero desde el exterior sólo se muestran como posiciones genéricas. En la vista interna se pueden dar variables de instancia prev y next a cada nodo v que indican, respectivamente, los nodos predecesor y sucesor de v (que de hecho, podrían ser los nodos centinelas delantero y trasero que indican el inicio y el final de la lista). A continuación, dada una posición p en S, se puede "desenvolver" a p para revelar el nodo v. Eso se logra ***convirtiendo*** la posición a un nodo. Una vez teniendo el nodo v se puede, por ejemplo, implementar el método before(p) regresando v.prev (a menos que v.prev sea el encabezado, en cuyo caso se produce un error). Por lo anterior, las posiciones, en una implementación con lista doblemente enlazada, se pueden soportar en una forma orientada a objetos sin tener más tiempo o espacio adicionales. Más aún, este método tiene la ventaja de ocultar al usuario los detalles de la implementación de la lista, lo cual ayuda a reusar el programa porque el usuario no sabe que los objetos de posición mostrados como parámetros y valores retornados en realidad son también objetos nodo.

En el fragmento de programa 5.3 se muestra una clase DNode de Java, para los nodos de una lista doblemente enlazada que implementa el TDA posición. Esta clase es parecida a la clase DLNode del fragmento de programa 4.13. Nótese que las variables de instancia prev y siguiente en esta clase son referencias privadas a otros objetos DNode. Al hacer privados esos campos se logra un grado de encapsulamiento porque se ocultan los detalles de cómo se implementan los nodos. Al usar esas variables de instancia sencillas, se puede usar la lista doblemente

enlazada para ejecutar todos los métodos del TDA lista en el tiempo $O(1)$. Por lo anterior, la lista doblemente enlazada es una implementación eficiente del TDA lista.

```
class DNode implements Position {
   private DNode prev, next;       // Referencias a los nodos antes y después
   private Object element;              // Elemento guardado en esta posición
   // Constructor
   public DNode(DNode newPrev, DNode newNext, Object elem) {
      prev= newPrev;
      next = newNext;
      element = elem;
   }
   // Método de la interfaz Posición
   public Object element() throws InvalidPositionException {
      if (( prev == null) && (next == null))
         throw new InvalidPositionException("¡La posición no está en la
         lista!");
      return element;
   }
   // Métodos accesores
   public DNode getNext() { return next; }
   public DNode getPrev() { return prev; }
   // Métodos de actualización
   public void setNext(DNode newNext) { next = newNext; }
   public void setPrev(DNode newPrev) { prev = newPrev; }
   public void setElement(Object newElement) { element = newElement; }
}
```

Fragmento de programa 5.3: La clase DNode realizando un nodo de una lista doblemente enlazada, e implementando la interfaz Position (TDA). Nótese cómo se usan los métodos accesor y actualizador para tener acceso, y actualizar las variables de instancia.

El método element de la clase DNode lanza una InvalidPositionException cuando sus campos prev y next se refieren al objeto nulo. El programa para esta excepción se muestra en el fragmento de programa 5.4.

```
// Una excepción para posiciones inválidas en el tiempo de ejecución
public class InvalidPositionException extends RuntimeException {
   public InvalidaPositionException (String err) {
      super(err)
   }
}
```

Fragmento de programa 5.4: Clase InvalidPositionException para posiciones que se usan en forma incorrecta en el tiempo de ejecución. En especial, nótese que esta clase extiende a la clase RuntimeException para condiciones de error que se presentan en la ejecución.

¿Cómo se podría implementar el método insertAfter(p,e), para insertar un elemento e después de la posición p? Se crea un nuevo nodo v para guardar al elemento e, se enlaza a v en su lugar en la lista y a continuación se actualizan las referencias next y prev de los dos nuevos vecinos de v. Este método se muestra como pseudocódigo en el fragmento de programa 5.5, y se ilustra en la figura 5.6. Recuérdese el uso de nodos centinelas delantero y trasero (sección 4.4.2), y nótese que este algoritmo trabaja aun cuando p sea la última posición real.

Algoritmo insertAfter(p,e):
Create a new node v
v.setElement(e)
v.setPrev(p) {enlazar v a su predecesor}
v.setNext(p.getNext()) {enlazar a v a su sucesor}
(p.getNext()).setPrev(v) {enlazar a v a el sucesor anterior de p}
p.setNext(v) {enlazar a p a su nuevo sucesor, v}
return v {la posición para el elemento e}

Fragmento de programa 5.5: Inserción de un elemento e después de la posición p en una lista enlazada.

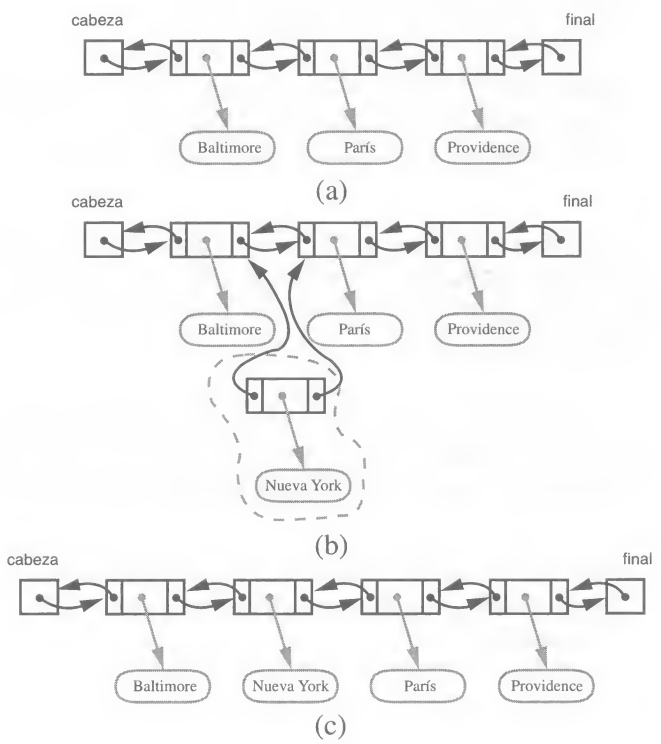

Figura 5.6: Adición de un nuevo nodo después de la posición para "Baltimore": (a) antes de la inserción; (b) creación del nodo v y enlazamiento del mismo; (c) después de la inserción.